Tiara: A Self-Stabilizing Deterministic Skip List

Thomas Clouser[1], Mikhail Nesterenko[1⋆], and Christian Scheideler[2]

[1] Deparment of Computer Science, Kent State University, Kent, OH, USA
[2] Institute of Computer Science, Technical University of Munich, Garching, Germany

Abstract. We present *Tiara* — a self-stabilizing peer-to-peer network maintenance algorithm. Tiara is truly deterministic which allows it to achieve exact performance bounds. Tiara allows logarithmic searches and topology updates. It is based on a novel *sparse 0-1 skip list*. We rigorously prove the algorithm correct in the shared register model. We then describe its extension to a ring and incorporation of crash tolerance.

1 Introduction

Due to the rise in popularity of peer-to-peer systems, dynamic overlay networks have recently received a lot of attention. An overlay network is a logical network formed by its participants across a wired or wireless domain. In open peer-to-peer systems, participants may frequently enter and leave the overlay network either voluntarily or due to failure. As peer-to-peer systems can contain millions of users, faults and inconsistencies should be regarded as the norm rather than an exception. Hence, overlay networks require mechanisms that continuously counter such disturbances. Simplistic ad hoc approaches that handle individual fault conditions do not adequately perform in case of unanticipated, complex or systemic failures. In practice many peer-to-peer systems, such as KaZaA, Bittorrent, Kademlia, use heuristic methods in order to maintain their topology. Moreover, solutions presented in research publications focus on constructing scalable and well-structured overlay networks in an efficient manner [1–9] while offering only ad hoc solutions to fault tolerance. For the overlay networks that are based on a sorted list or ring (e.g., [2, 3, 5, 9]), recovery can be achieved as long as this base structure can be maintained. However, jointly maintaining such list and the complete structure is rather tricky.

One can argue that if nodes are randomly distributed, a sorted list or ring with a sufficient number of redundant connections will not disintegrate with high probability. However, it is not clear whether practical systems always satisfy such randomization assumption. In addition, the problem of generating high-quality trusted random numbers in a peer-to-peer systems is far from trivial. Moreover, it is known that an adversary can quickly degrade the randomness of the peer-to-peer system even if perfectly random numbers are reliably generated [10]. Thus, some researchers [11, 12] argue that overlay network architects need to consider holistic approaches to fault tolerance and recovery, such as self-stabilization. In

⋆ This research is supported in part by NSF Career award CNS-0347485.

this paper we present Tiara. To the best of our knowledge, Tiara is the first self-stabilizing skip-list based overlay network algorithm that supports logarithmic searches and updates.

Related literature. Several algorithms presented in the literature focus on stabilizing parts of overlay networks. Onus et al. [12] present several high-atomicity solutions to linearizing an overlay network. Shaker and Reeves [13] describe a distributed algorithm for forming a directed ring network topology. Hérault et al. [14] describe a spanning tree formation algorithm for overlay networks. Cramer and Fuhrmann [15] show that ISPRP — a ring-based overlay network is, in certain cases, self-stabilizing. Caron et al. [16] describe a snap-stabilizing prefix tree for peer-to-peer systems. Bianchi et al. [17] present a stabilizing search tree for overlay networks optimized for content filters.

Several randomized overlay network algorithms have also been proposed. Dolev and Kat [18] introduce the HyperTree and use it as a basis for their self-stabilizing peer-to-peer system. Dolev et al. [19] describe a self-stabilizing intrusion-tolerant overlay network.

Pugh [20] introduce skip lists as an alternative to balanced tree structures. Munro et al. [21] describe a deterministic algorithm for skip list construction. Awerbuch and Scheideler [3], Aspnes and Shah [2], and Harvey et al. [5] extend the randomized skip list to distributed environments. Harvey and Munro [22] present a deterministic distributed skip list.

Our contribution. In this paper we present Tiara. It stabilizes a novel 0-1 distributed skip list. Specifically, we demonstrate a self-stabilizing algorithm for a sorted list and then show how to extend it to a self-stabilizing algorithm for a skip list. Tiara can construct these structures without any knowledge of global network parameters such as the number of nodes in the system, each node utilizes only the information available to its immediate neighbors. Moreover, Tiara preserves network connectivity so long as the initial network is connected. That is, Tiara reconstructs the connectivity of the base sorted list on the basis of skip list links. We rigorously prove Tiara correct in an asynchronous communication register based model.We describe how Tiara can be extended to a ring structure and how it can incorporate crash resistance.

Organization of the paper. First, we introduce our computational model. Then, we describe a self-stabilizing algorithm for the sorted list and formally prove it correct. We then extend it to a self-stabilizing algorithm for Tiara discuss various extensions and efficiency improvements. We complete the paper with future research directions and open problems.

2 Model

A peer-to-peer system consists of a set N of processes. Each process has a unique integer identifier. A process contains a set of variables and actions. An action has

the form $\langle name \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$. *name* is a label, *guard* is a Boolean predicate over the variables of the process and *command* is a sequence assigning new values to the variables of the process. For each pair of processes a and b, we define a Boolean variable (a, b) that is shared among them. Two processes a and b are *neighbors* if this variable is **true**. The *neighborhood* of a process a is defined as the set of all of its neighbors. Sets of neighbors may be maintained on different *levels*. A neighborhood of process a at level i is denoted and denoted $a.i.NB$. The *right neighborhood* of a, denoted $a.i.R$, is the set of neighbors of a with identifiers larger than a. That is, $a.i.R \equiv \{b : b \in a.i.NB : b > a\}$. Similarly, the *left neighborhood* of a, denoted $a.i.L$, are a's neighbors with smaller identifiers. That is, $a.i.L \equiv \{b : b \in b.i.NB : b < a\}$. Naturally, the union of $a.i.R$ and $a.i.L$ is $a.i.NB$.

When describing a link we always state the smaller identifier first. That is, a is less than b in (a, b). Two processes a and b are *consequent* if there is no process c whose identifier is between a and b. That is, $\mathbf{cnsq}(a, b) \equiv (\forall c :: (c < a) \vee (b < c))$. The *length* of a link (a, b) is the number of processes c such that $a < c < b$. By this definition the length of a link that connects consequent processes is zero.

A *system state* is an assignment of a value to the variables of each process. An action is *enabled* in some state if its guard is **true** at this state. A *computation* is a maximal fair sequence of states such that for each state s_i, the next state s_{i+1} is obtained by executing the command of an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. The execution of a single action is a *step*. Maximality of a computation means that the computation is infinite or it terminates in a state where none of the actions are enabled. Such state is a *fixpoint*. In a computation the action execution is *weakly fair*. That is, if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often. This defines an *asynchronous* program execution model.

A state *conforms* to a predicate if this predicate is **true** in this state; otherwise the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**. Let T and U be predicates over the state of the program. Predicate T is *closed* with respect to the program actions if every state of the computation that starts in a state conforming to T also conforms to T. Predicate T *converges* to U if T and U are closed and any computation starting from a state conforming to T contains a state conforming to U. The program *stabilizes* to T if **true** converges to T. Since we will focus on self-stabilizing algorithms for overlay networks, and self-stabilization is only possible for overlay networks that are initially connected, we identify with **true** any state where the graph is connected.

While most of our program model is fairly conventional, we would like to draw the reader's attention to our way of modelling overlay network link management. If one process updates its neighborhood, the change affects the neighbors of other processes. For example, if process a adds b to its neighborhood by creating a link (a, b), this also means that a is atomically added to b's neighborhood. On the

other hand, if a removes b from its neighborhood, then also a is removed from b's neighborhood.

3 Core Tiara Description, Correctness Proof and Complexity Estimate

In its core, Tiara contains two components: the bottom component (b-Tiara) that maintains the processes at the lowest level in sorted order and the skip-list component (s-Tiara) that constructs the higher levels of Tiara. These components are interdependent. s-Tiara relies on b-Tiara to sort the lowest level, while s-Tiara may append links to the bottom level to preserve the connectivity of the system.

We present the components and prove them correct bottom up starting with b-Tiara. However, the presentation of b-Tiara is divided into two parts: the growing and trimming. We prove the stabilization of the growing part first as the stabilization of s-Tiara depends on its correct operation. We prove the stabilization of the trimming part last as it depends on the stabilization of s-Tiara.

process u
variables
$u.0.NB$ — set of neighbor processes of u.
shortcuts
$u.0.L \equiv \{z : z \in u.0.NB : z < u\},\ u.0.R \equiv \{z : z \in u.0.NB : z > u\}$
actions
grow right: $(s \in u.0.R) \wedge (t \in s.0.L) \wedge (t \notin u.0.NB) \longrightarrow$
$u.0.NB := u.0.NB \cup \{t\}$
trim right: $(s, t \in u.0.R) \wedge (t \in s.0.L) \wedge (\forall z : z \in u.0.R : z \leq s) \wedge (\forall z : z \in s.0.L : z \geq u) \longrightarrow$
$u.0.NB := u.0.NB/\{s\}$
grow left and *trim left* are similar

Fig. 1. The bottom component of Tiara (b-Tiara).

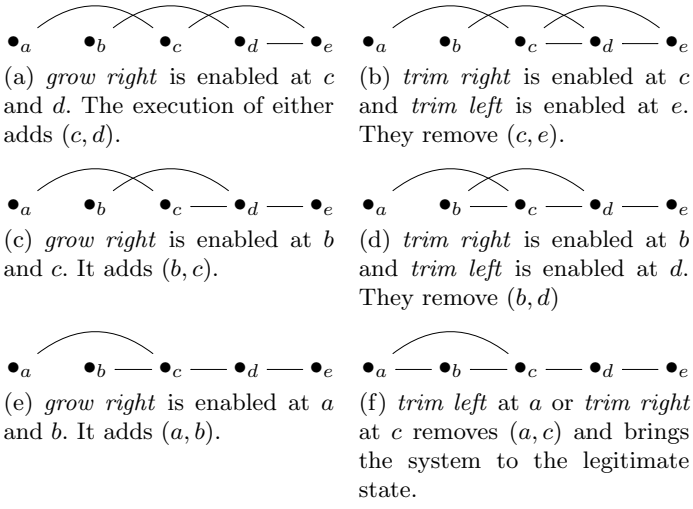
3.1 The Bottom Component of Tiara (b-Tiara) and Stabilization of Grow

Description. The objective of b-Tiara is to transform the system into a linear graph with the processes sorted according to their identifiers. The algorithm for b-Tiara is shown in Fig. 1. The only variables that b-Tiara manipulates are the neighbor sets for each process u — $u.0.NB$. The *right neighborhood* of u, denoted $u.0.R$ is a subset of $u.0.NB$ with the identifiers greater than u. Since $u.0.R$ can be computed from $u.0.NB$ as necessary, $u.0.R$ is not an independent variable but a convenient shortcut. The *left neighborhood* $u.0.L$ is defined similarly.

Each process u has two pairs of actions: *grow* and *trim* that operate to the right and to the left of u. Action *grow right* is enabled if u discovers that its

right neighbor s has a left neighbor t that is not a neighbor of u. In this case u adds t to its neighborhood. That is, u adds a link (u, t) to the graph. Even though u is the left neighbor of s, t may be either to the left or to the right of u. That is $t < u$ or $t > u$. Regardless of this relation, u connects to t. Action *grow left* operates similarly in the opposite direction.

Action *trim right* eliminates extraneous links from the graph. This action removes link (u, s) if u has a neighbor s that satisfies the following properties. The guard for *trim right* stipulates that there has to be another process t that is a neighbor of both u and s. Hence, if (u, s) is removed the connectivity of the graph is preserved. Also, all right neighbors of u must be smaller than or equal to s and all left neighbors of s are greater than or equal to u. The latter condition is necessary to break symmetry and prevent continuous growing and trimming of the same link. Action *trim left* operates similarly in the reverse direction. We show an example operation of b-Tiara in Fig. 2.



(a) *grow right* is enabled at c and d. The execution of either adds (c, d).

(b) *trim right* is enabled at c and *trim left* is enabled at e. They remove (c, e).

(c) *grow right* is enabled at b and c. It adds (b, c).

(d) *trim right* is enabled at b and *trim left* is enabled at d. They remove (b, d)

(e) *grow right* is enabled at a and b. It adds (a, b).

(f) *trim left* at a or *trim right* at c removes (a, c) and brings the system to the legitimate state.

Fig. 2. Example computation of *b-Tiara*. The processes are listed in increasing order of their identifiers.

Correctness proof. Denote $B(N)$ the graph that is induced by the processes of the system and the links of b-Tiara. We define the following predicate: $\mathcal{GI} \equiv (\forall a, b \in N :: \mathbf{cnsq}(a, b) \Rightarrow \exists(a, b))$. That is, $\mathcal{GI}$ states that two consequent processes are also neighbors.

Lemma 1. *If a computation of b-Tiara starts from a state where $B(N)$ is connected, it is connected in every state of this computation.*

Proof: The actions of b-Tiara do not disconnect $B(N)$. Indeed, the actions that remove links are *trim right* and *trim left*. Consider *trim right*. It removes a

link (a, b) if there exists a node c such that there are links (a, c) and (c, b). Thus, the removal of (a, b) does not disconnect the graph. The argument for *trim left* is similar. □

Lemma 2. *If a computation of b-Tiara starts from a state where $B(N)$ is connected, b-Tiara stabilizes to $\mathcal{GI}$.*

Proof: To prove the lemma we need to show that (i) $\mathcal{GI}$ is closed under the execution of the actions of b-Tiara and (ii) regardless of the initial state, every computation contains a state satisfying $\mathcal{GI}$. Let us consider closure first. The *grow* actions may not violate $\mathcal{GI}$ as they only add links. The *trim* action may affect $\mathcal{GI}$ by disconnecting two processes a and b. However, *trim right*, which removes link (a, b), is only enabled at process a if there is a process c such that $a < c < b$. Therefore, if a and b are consequent, *trim right* is disabled. The reasoning is similar for *trim left*. Hence the closure.

To show convergence, let us assume that there are two consequent processes a and b that are not neighbors. That is $b \notin a.0.NB$. Since the graph itself is connected, there is a path ρ between a and b. If there are multiple paths, we shall consider the shortest one. Let the length of ρ be the sum of the lengths of its constituent links. The execution of a *trim* action does not change the length of ρ. The execution of any of the *grow* actions does not increase the length of ρ. Path ρ must contain at least one segment d, e, f such that both d and f are either smaller than e or larger than e. In this case *grow right*, or respectively, *grow left*, is enabled in both d and f. The execution of this action decreases the length of the path. Hence, throughout the computation, the length of ρ decreases until it is zero and a and b are neighbors. The lemma follows. □

3.2 The Skip List Component of Tiara (s-Tiara)

Description. The objective of s-Tiara is to establish a skip list on top of the linearized graph created by b-Tiara. The structure maintained by s-Tiara is a *sparse 0-1 skip list*. At each level i, node u maintains a set of neighbors $u.i.NB$. Out of this set, the rightmost and leftmost neighbors are defined as right and left skip links: $u.i.rs$ and $u.i.ls$. A node may not have a right or left skip link at some level if it is on either end of the list.

We denote right and left skip list neighbors of u at level $i - 1$ as v and x respectively. Nodes w and y are respectively right and left neighbors of v and x at the same level. We illustrate this notation in Fig. 3 as we will be using it extensively throughout the correctness proof of the algorithm.

If both nodes u and v exist at level i and $u.i.rs = v$ then this link is 0-*skip link*. If u and w exist at level i and $u.i.rs = w$, then this link is a 1-*skip link*. A process that exists at level $i - 1$ is *up* if it also exists at level i, it is *down* otherwise. If a process that 1-skip link spans is down it is a *cage*. For example u, v and w form a cage if $u.i.rs$ links to w and v is down. The middle process is *inside* the cage. Refer to Fig. 4 for the illustration of the concept of a cage. The sparse 0-1 skip list has two rules of organization. First, all links are either 0 or

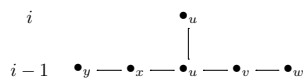
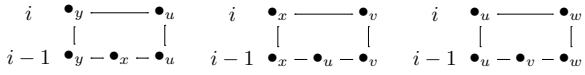


Fig. 3. Aliases for neighbors of u in *s-Tiara*. $v \equiv u.(i-1).rs$, $w \equiv v.(i-1).rs$, $x \equiv u.(i-1).ls$, and $y \equiv x.(i-1).ls$, where $u.i.rs$ and $u.i.ls$ are right and left skip-list neighbors of u at level i, respectively.

1 skip links. Second, if a node is on level i and it is not on the end of the list on level $i-1$ then at least one of its links is a 1 skip link.



(a) u is adjacent to the cage on the left.

(b) u is inside the cage.

(c) u is adjacent to the cage on the right.

Fig. 4. Possible cages with respect to node u.

The the algorithm is shown in Fig. 5. As before, to simplify the presentation we introduce a few shortcuts. Sets $u.i.R$ and $u.i.L$ are the subsets of $u.i.NB$ that contain the identifiers of u's neighbors with respectively higher and lower identifiers than u. We define $u.i.rs$ to be the neighbor with the link of the smallest length among $u.i.R$. To put another way, $u.i.rs$ connects to u's right neighbor with the smallest identifier. Note that $u.i.rs$ is $\perp$ if $u.i.R$ is empty. Shortcut $u.i.ls$ is defined similarly.

Predicate **exists**(z, i) is **true** if node z is present at all and if $z.i.NB$ is not empty. Node u may read only its immediate neighbor states. Thus, u may only invoke **exists** on its neighbors and itself. Observe that **exists** is defined to return **false** if it is invoked on a non-existent node. For example, if u is at the right end of the list at level i and u invokes **exists**$(u.i.rs, i)$. In this case **exists**$(u.i.rs, i)$ returns **false**. Predicate **valid**(u, i) captures the correct state of the system. Specifically, it states that if u exists at level i then the length of the skip links should not be more than 1 and either x or v does not exist at level i. The latter condition guarantees that at least one link of u is a 1 skip link.

The actions of s-Tiara are as follows. Action *upgrade right* establishes a link to w at level i if v is not up. That is, this link is a 1 skip link. If u is not up, *upgrade right* brings u up to level i. Action *upgrade left* operates similarly in the opposite direction. Actions *bridge right* and *left* establish 0 skip links if both nodes being connected are up. Action *prune* eliminates the links other than $u.i.rs$ and $u.i.ls$ from $u.i.NB$. In case the links are not 0 or 1 skip, action *downgrade right* completely removes the right neighborhood of u. Action *downgrade left* operates similarly. And the last action *downgrade center* eliminates three

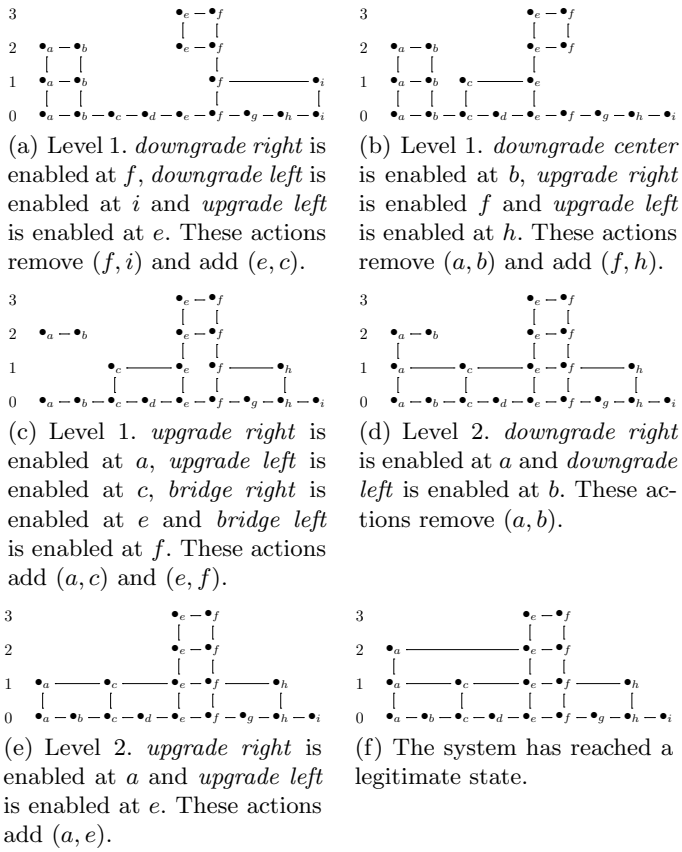
process u
parameter $i \geq 0$: **integer** — level of the skip list
variables
$u.i.NB$ — set of neighbor processes of u at level i
shortcuts
$v \equiv u.(i-1).rs,\ w \equiv v.(i-1).rs,\ x \equiv u.(i-1).ls,\ y \equiv x.(i-1).ls$
$u.i.R \equiv \{z : z \in u.i.NB : z > u\},\ u.i.L \equiv \{z : z \in u.i.NB : z < u\}$
$u.i.rs \equiv \begin{cases} (s : s \in u.i.R : (\forall t : t \in u.i.R : t \geq y)), & \text{if } u.i.R \neq \varnothing \\ \bot, & \text{otherwise} \end{cases}$
$u.i.ls$ is defined similarly
$\mathbf{exists}(z, i) \equiv ((z \neq \bot) \wedge (z.i.NB \neq \varnothing))$
$\mathbf{valid}(u, i) \equiv ((((u.i.ls = y) \vee (u.i.ls = x) \vee (u.i.ls = \bot)) \wedge (u.i.rs = w)) \vee$
$(((u.i.rs = v) \vee (u.i.rs = w) \vee (u.i.rs = \bot)) \wedge (u.i.ls = y)) \vee$
$((u.i.ls = \bot) \wedge (u.i.rs = \bot)) \vee$
$\neg(\mathbf{exists}(x, i) \wedge \mathbf{exists}(u, i) \wedge \mathbf{exists}(v, i)))$
actions for $i > 0$
upgrade right: $\mathbf{valid}(u, i) \wedge \neg\mathbf{exists}(v, i) \wedge (v \neq \bot) \wedge (w \neq \bot) \wedge (u.i.rs \neq w) \longrightarrow$
$u.i.NB := u.i.NB \cup \{w\}$
upgrade left is similar
bridge right: $\mathbf{valid}(u, i) \wedge \mathbf{exists}(u, i) \wedge \mathbf{exists}(v, i) \wedge (u.i.rs \neq v) \longrightarrow$
$u.i.NB := u.i.NB \cup \{v\}$
bridge left is similar
prune: $\mathbf{valid}(u, i) \wedge \mathbf{exists}(u, i) \wedge (u.i.NB \neq \{u.i.rs, u.i.ls\}) \longrightarrow$
$u.0.NB := u.0.NB \cup u.i.NB / \{u.i.rs, u.i.ls\},$
$u.i.NB := \{u.i.rs, u.i.ls\}$
downgrade right: $\neg\mathbf{valid}(u, i) \wedge \neg((u.i.rs = v) \vee (u.i.rs = w) \vee (u.i.rs = \bot)) \longrightarrow$
$u.0.NB := u.0.NB \cup u.i.R,$
$u.i.R := \varnothing$
downgrade left is similar
downgrade center: $\neg\mathbf{valid}(u, i) \wedge \mathbf{exists}(x, i) \wedge \mathbf{exists}(u, i) \wedge \mathbf{exists}(v, i) \longrightarrow$
$u.0.NB := u.0.NB \cup u.i.NB,$
$u.i.NB := \varnothing$

Fig. 5. The skip list component of Tiara (s-Tiara).

consecutive up nodes. This ensures that there could not be two consecutive 0 skip links. An example computation of s-Tiara is shown in Fig. 6.

Correctness proof. Our proof proceeds as follows. We state five predicates on the level i of s-Tiara. In the sequence of lemmas we show that if the lower levels of s-Tiara have stabilized, then level i of s-Tiara stabilizes to these predicates. The conjunction of these predicates implies the stabilization of level i of s-Tiara. We then use this fact as an inductive step in the convergence proof of stabilization of s-Tiara.

Before proceeding with the proof, we introduce notation and terminology we are going to use. Denote $S(N)$ the graph induced by the processes of the system as well as the links of b-Tiara and s-Tiara. Throughout the discussion we



(a) Level 1. *downgrade right* is enabled at f, *downgrade left* is enabled at i and *upgrade left* is enabled at e. These actions remove (f, i) and add (e, c).

(b) Level 1. *downgrade center* is enabled at b, *upgrade right* is enabled f and *upgrade left* is enabled at h. These actions remove (a, b) and add (f, h).

(c) Level 1. *upgrade right* is enabled at a, *upgrade left* is enabled at c, *bridge right* is enabled at e and *bridge left* is enabled at f. These actions add (a, c) and (e, f).

(d) Level 2. *downgrade right* is enabled at a and *downgrade left* is enabled at b. These actions remove (a, b).

(e) Level 2. *upgrade right* is enabled at a and *upgrade left* is enabled at e. These actions add (a, e).

(f) The system has reached a legitimate state.

Fig. 6. *s-Tiara*. We list the processes in the increasing order of their identifiers. *b-Tiara* has stabilized to $\mathcal{GI}$. In each state we only mention the enabled actions that are relevant to the discussion. We do not illustrate the operation of *prune*.

consider process u and its neighbors as defined in the description of s-Tiara. A node u is *middle* at level i if it has both left and right neighbors as well at least one two hop neighbor. That is, $\mathbf{middle}(u, i) \equiv (\mathbf{exists}(v, i-1) \wedge \mathbf{exists}(x, i-1) \wedge (\mathbf{exists}(y, i-1) \vee \mathbf{exists}(w, i-1)))$.

Below are the predicates to which s-Tiara stabilizes. Predicate $\mathbf{good\_links}.i$ states that process u connects to processes at most two hops away. Predicate $\mathbf{one\_links}.i$ enforces the rules of 0-1 skip list. Specifically, it stipulates that u should either be inside the cage or should have adjacent cages to the left or to the right. Predicates $\mathbf{zero\_left\_links}.i$ and $\mathbf{zero\_left\_links}.i$ ensure that the 0-links are in place. That is, the processes that are consequent at level $i-1$ and are up, are also connected at level i. Predicate $\mathbf{only\_good\_links}.i$ states that the neighborhood of u does not have links other than rs and ls.

$$\textbf{good\_links}.i \equiv (\forall u :: \neg\textbf{exists}(u.i) \vee ((u.i.rs = v) \vee (u.i.rs = w) \vee (u.i.rs = \perp) \wedge ((u.i.ls = y) \vee (u.i.rs = x) \vee (u.i.ls = \perp))$$

$$\textbf{one\_links}.i \equiv (\forall u : \textbf{middle}(u, i) : (\neg\textbf{exists}(u, i) \wedge (x.i.rs = v) \wedge (v.i.ls = x)) \vee (\neg\textbf{exists}(v, i) \wedge (\neg\textbf{exists}(w, i-1) \vee (u.i.rs = w))) \vee (\neg\textbf{exists}(x, i) \wedge (\neg\textbf{exists}(y, i-1) \vee (u.i.ls = y))))$$

$$\textbf{zero\_right\_links}.i \equiv (\forall u :: \neg\textbf{exists}(u.i) \vee \neg\textbf{exists}(v.i) \vee (u.i.rs = v))$$

$$\textbf{zero\_left\_links}.i \equiv (\forall u :: \neg\textbf{exists}(u.i) \vee \neg\textbf{exists}(x.i) \vee (u.i.ls = x))$$

$$\textbf{only\_good\_links}.i \equiv (\forall u :: \neg\textbf{exists}(u.i) \vee (u.i.NB = \{u.i.rs, u.i.ls\}))$$

Lemma 3. *Assuming that neighbor relations at level* $i-1$ *do not change throughout the computation, s-Tiara stabilizes to* **good_links**.i

Proof: In proving this and consequent lemmas we show a stronger property of closure and convergence of the predicate for a particular process u. This implies the stabilization of the predicate for all u at the specified level.

Let us show closure first. The topology at level $i-1$ does not change. Hence once $u.i.rs$ points to one or two hop neighbors v or w, the neighbor's relative positions do not change. Similar argument applies to $u.i.ls$. Let us consider the actions and how they affect **good_links**.i. Let us start with the actions of u. Actions *upgrade right* and *bridge right* do not violate the predicate since they set $u.i.rs$ to respectively w and v. Similar argument applies to *upgrade left* and *bridge left*. Action *prune* does not affect the predicate since it does not modify either $u.i.rs$ or $u.i.ls$. Neither do *downgrade right* and *downgrade left* since they respectively set $u.i.rs$ and $u.i.ls$ to $\perp$. Action *downgrade center* removes u from level i altogether and hence cannot violate the predicate. The nodes further than two hops away never connect to u. Hence the actions of other nodes cannot violate the predicate either.

Let us now address convergence. The predicate can be violated only if u is up. It is violated if either $u.i.rs$ or $u.i.ls$ points to a node other than u's one or two-hop neighbors. In this case either *downgrade right* or *downgrade left* are enabled that bring the links in compliance with the predicate. □

Lemma 4. *Assuming that neighbor relations at level* $i-1$ *do not change throughout the computation and* **good_links**.i *is satisfied, s-Tiara stabilizes to* **one_links**.i

Proof: As a first step, we would like to make the following observation: once a cage is formed, it is never destroyed. For example, assume that u, v and w form a cage. The actions of u, and, similarly, w do not affect this link. Also, if v is down, the only actions it can use to come up is *upgrade right* or *upgrade left*. However, both are disabled since u and v are up. This observation guarantees the closure of **one_links**.i.

Let us discuss convergence. Assume that u is down. We consider two cases: u is initially down and u is initially up and never goes down. If u is down, the

only way, u can come up is through execution of *upgrade right* or *upgrade left* at u, w or y. In all cases cages adjacent to u are formed and the predicate is satisfied. If u is down, then *upgrade right* is enabled in x and *upgrade left* in v. Thus if u does not come up, then x or v execute these *upgrade* actions. In which case a cage is formed with u inside. This satisfies the predicate as well.

Assume that u is up. If it ever goes down, the foregoing discussion applies. The only remaining case is if u stays up for the remainder of the computation. Throughout a computation of b-Tiara a node can come up only once. Indeed, a node comes up only if it forms a cage. Since a cage is never destroyed, the node never goes down. This means that a node can go down only once. Let us consider the state of the computation where u's neighbors x and v do not change their up and down position. Both x and v cannot be simultaneously up in this state, as it enables *downgrade center* at u. The execution of this action brings u down. However, we assumed that u stays up for the remainder of the computation. Thus, either x or v are down. Assume, without loss of generality, that v is down. If w does not exist at level $i-1$, **one_links**.i is satisfied. Assume that w exists. If link $u.i.rs = w$ is present, **one_links**.i is also satisfied. However, if it is not present, then *upgrade right* is enabled in u. Its execution establishes the link, forms a cage and satisfies the predicate. □

Lemma 5. *Assuming that neighbor relations at level* $i-1$ *do not change throughout the computation and* **good_links**.i *as well as* **one_links**.i *are satisfied, s-Tiara stabilizes to* **zero_left_links**.i *and* **zero_right_links**.i

Proof: We prove the lemma for **zero_right_links**.i only. The proof for the other predicate is similar. Let us argue closure. If **one_links**.i is satisfied processes do not go up or down. Thus, the only actions that can be enabled are *bridge* and *prune*. The execution of either action maintains the validity of **zero_left_links**.i. Hence the closure.

Let us address convergence. The predicate is violated only if the neighbor processes u and v are both up and they do not have a link at level i. If **one_links**.i is satisfied, u forms a cage to its left, while v forms a cage to its right. Recall that the cages are never destroyed. In this case u has *bridge right* while v has *bridge left* enabled. When either action is executed the predicate is satisfied. □

Lemma 6. *Assuming that neighbor relation at level* $i-1$ *does not change throughout the computation and* **good_links**.i, **one_links**.i, **zero_right_links**.i *as well as* **zero_left_links**.i *are satisfied, s-Tiara stabilizes to* **only_good_links**.i

Proof: (outline) The satisfaction of **good_links**.i, **one_links**.i, **zero_right_links**.i and **zero_left_links**.i leaves only one possible action enabled — *prune*. In this case there are links in $u.i.NB$ besides $u.i.rs$ and $u.i.ls$ and they are moved to $u.0.NB$. □

Lemma 7. *If a computation of Tiara starts from a state where* $S(N)$ *is connected, this computation contains a state where* $B(N)$ *is connected.*

Proof: The non-trivial case is where $S(N)$ is connected while $B(N)$ is not. That is, the overall graph connectivity is achieved through the links at the higher levels of Tiara. Let X and Y be two graph components of $B(N)$ such that they are connected in $S(N)$. Let $i > 0$ be the lowest level where X and Y are connected. Assume, without loss of generality that there is a pair of processes $a \in X$ and $b \in Y$, such that $a.i.rs = b$. In this case *downgrade right* is enabled at a. The execution of *downgrade right* connects X And Y in $B(N)$. The lemma follows. □

Define

$$\mathcal{SI} \equiv (\forall i : i > 0 : \mathbf{good\_links}.i \wedge \mathbf{one\_links}.i \wedge \mathbf{zero\_right\_links}.i \wedge \mathbf{zero\_left\_links}.i \wedge \mathbf{only\_good\_links}.i)$$

Lemma 8. *Tiara stabilizes to $\mathcal{SI}$.*

Proof: According to Lemma 7, every computation contains a state where $B(N)$ is connected. Due to Lemma 2, if $B(N)$ is connected, b-Tiara stabilizes to $\mathcal{GI}$. The remainder of the proof is by induction on the levels of s-Tiara. If $B(N)$ is connected and $\mathcal{GI}$ is satisfied the topology of the level 0 does not change. Hence, the requisite five predicates are vacuously satisfied. Assume that these predicates are satisfied for all levels $i - 1$. Once the predicates are satisfied, none of the actions for processes at level $i - 1$ are enabled. This means that the topology at this level does not change. Applying Lemmas 3, 4 5 and 6 in sequence we establish that the five predicates are satisfied at level i. Hence the lemma. □

3.3 Stabilization of Trim in b-Tiara

Link (a, b) is *independent* if there exists no link (c, d) different from (a, b) such that $c \leq a$ and $b \leq d$. Consider an arrangement where the nodes are positioned in the increasing order of their identifiers.

Lemma 9. *If a computation of b-Tiara that starts in a state where the graph is connected and contains an independent link of non-zero length, this computation also contains a suffix of states without this link.*

Proof: Let (a, b) be an independent link of non-zero length. None of the *grow* actions create independent links. The only action that makes a link independent is a *trim* of another independent link. Thus, if an independent link is deleted, it is never added. Thus, to prove the lemma it is sufficient to show that (a, b) is eventually deleted.

Link (a, b) is non-zero length. This means that the node c consequent to a is not the same as b. In other words $a < c < b$. b-Tiara stabilizes to $\mathcal{GI}$ which ensures that a and c are connected. If c and b are not connected, both of them have a *grow* action enabled that connects them. Observe that (a, b) is independent. This means that all the right neighbors of a are to the left of b

and all the left neighbors of b are to the right of a. Moreover, we just showed that there exists a node c such that $a < c < b$ and there are links $c \in a.R$ and $c \in b.L$. This means that *trim right* is enabled at a and *trim left* is enabled at b. The execution of either action deletes (a, b). □

We define the following predicate: $\mathcal{TI} \equiv (\forall a, b \in N :: \exists(a, b) \Rightarrow \mathbf{cnsq}(a, b))$

Lemma 10. *If Tiara starts in a state where it satisfies $\mathcal{GI}$ and $\mathcal{SI}$, then it stabilizes to $\mathcal{TI}$*

Proof: (outline) The conjunct of $\mathcal{GI}$ and $\mathcal{TI}$ is closed under the execution of b-Tiara. Note also that if $\mathcal{GI}$ and $\mathcal{SI}$ are satisfied, then the actions s-Tiara are disabled. Hence the closure of $\mathcal{TI}$.

Let us consider convergence. Since the actions of s-Tiara are disabled, they do not add links to $B(N)$. If $\mathcal{TI}$ does not hold, then there is at least one independent link of non-zero length. If the graph is connected the *grow* actions never create an independent link. Consider a computation of b-Tiara that starts in an illegitimate state. Let l be the length of the longest independent link. Since the state is not legitimate, $l > 0$. According to previous discussion, new links of length l do not appear. Let (a, b) be the independent link of length l. According to Lemma 9, (a, b) is eventually removed. Thus, all links of length l are eventually removed. The lemma can be easily proven by induction on l. □

The discussion in this section culminates in the following theorem.

Theorem 1. *Tiara stabilizes to the conjunction of $\mathcal{GI}$, $\mathcal{SI}$ and $\mathcal{TI}$.*

4 Tiara Usage, Implementation and Extensions

Searches. Tiara maintains a skip list [20, 21] which is equivalent to a distributed balanced search tree. Hence the searches in Tiara proceed similar to searches in such trees. Let b be a right neighbor of a at some level i of Tiara. The *right interval* of a, denoted $[a, b)$, is the range of identifiers between a and b. *Left interval* is defined similarly. If a does not have a right neighbor, its interval is not finite. That is, a's interval contains all process identifiers greater than a. Similarly, if a lacks left neighbor it's interval is infinite on the left. Thus in any level, the collection of intervals contains the complete range of identifiers.

Suppose a, c and b are consequent at level $i - 1$ of Tiara and a and b are consequent at level i. That is c is in the cage. Since the identifiers are sorted, c belongs to the interval $[a, b)$. If a node is down, then one if its neighbors is up. Thus a client process that has a pointer to a node in Tiara and wishing to advance up the skip list only needs to examine the node's neighbors.

Assuming that a client process connects to an arbitrary node in Tiara, the search proceeds first upward then downward in the skip list. In the upward phase, the client is moving up the list looking for the node whose interval contains the identity. Since every level contains the complete id-range, this phase terminates.

Once the range is found, the client advances downward evaluating the cages it encounters to narrow the search range. This procedure continues until the desired node x is located or it is established that x belongs to the interval of the consequent nodes at the bottom level. The latter case means that x is not present in the system. There are $O(log|N|)$ levels in Tiara. Thus, the upward and the downward phases take $O(log|N|)$ number of steps.

Joins and leaves. We assume that each process has two read-only Boolean variables maintained by the environment: *join* and *leave*. Since the variables are read-only, stabilization of their operation is the responsibility of the environment. Let us consider join operation first. The joining node x connects to an arbitrary node of the network. The variable *join* is set to **true**. We assume that the environment may only set *join* to **false** after the node successfully inserts itself at the bottom level of Tiara. The joining node executes a search to find the bottom level interval $[a, b)$ to which it belongs. Then, x makes a and b its right and left neighbors respectively. After a and b discover the presence of a node whose *join* is set to **true**, they remove link (a, b). Then, the upper levels of Tiara adjust. The insertion of the node at the bottom level entails at most a constant number of steps at each level of Tiara. Since the search takes at most $O(log|N|)$ steps, the total number of steps required for node join is also in $O(log|N|)$.

Let us discuss the leave operation. The environment sets *leave* to **true** to indicate that the node x requests disconnect. We assume that *leave* cannot be set when *join* is set and it cannot be set back to **false** until the node disconnects. When the right and left neighbors of x notice that the *leave* of x is set to **true**, the neighbors add a link bypassing x at the bottom level. Node x can then disconnect. The higher levels of Tiara execute the regular Tiara actions to accommodate the missing node. At most a constant number of adjustment steps is required at each level. Hence the total number of steps required for the node to leave Tiara is in $O(log|N|)$.

Crash resistance. Tiara can be separated into disconnected components by the crash of even a single process. Tiara can be fortified against separation due to crashes in the following manner. At the bottom, each process maintains a crash-redundancy link to its right neighbor's neighbor. That is, the bottom level list becomes doubly connected. Thus, it can tolerate a single crash. The crash tolerance can be further improved by adding similar links to more distant processes. In an asynchronous model there is no reliable way to distinguish a crashed process from a slow one [23]. Thus, to accomplish this, the processes need to be equipped with failure detectors [24, 25]. A failure detector alerts the process if its neighbor crashes. Then, Tiara stabilizes to a legitimate state corresponding to the system without the crashed process.

Extension to ring. Tiara can be extended to a ring structure similar to Chord [9]. The idea is as follows. For b-Tiara, as well as for each level of s-Tiara, the lowest id-process needs to add a special wraparound link to the highest-id

process. This wraparound link maintenance is carried out by the process without left neighbors. After b-Tiara and s-Tiara stabilize, the lowest-id process at each level is the only such process. The highest-id process at each level is the only process without right neighbors.

Once the process determines that it has no left neighbors it starts positioning the wraparound link. Essentially, the process continues to move the link to a right neighbor of the destination of the link. Note that this movement stops once the wraparound link reaches the highest-id process at that level. If the maintainer of the wraparound link determines that it has left neighbors, it destroys its wraparound link. Refer to a technical report [26] for a detailed description of this extension.

Other improvements. There is a number of modifications to Tiara that make it more efficient and applicable. At each level of Tiara, up to two out of three nodes may be promoted to the next level. Although the number of levels is logarithmic with respect to the system size, it may still be relatively large. The number of levels may be decreased by modifying Tiara to promote fewer nodes. For example, we can allow the nodes at level i to skip up to two or three neighbors at level $i-1$. This would require for each node to maintain data about its extended neighborhood.

The *grow* operation of b-Tiara may force a process to acquire up to $O(|N|)$ neighbors during stabilization. This may require devoting extensive memory resources of each node to neighborhood maintenance. A simple way to mitigate it is to execute *trim* operations before *grow*. That is, if a process finds that it has both *trim* and *grow* actions enabled. It executes *trim*. Care must be taken to ensure that action execution is still weakly fair.

5 Future Work

We presented Tiara — a first deterministic self-stabilizing peer-to-peer system with a logarithmic diameter. It provides a blueprint for a realistic system. We envision several directions of extending this work: further efficiency improvements, such as keeping the runtime and the degree of the self-stabilization process low, and adding features required by practical systems. One interesting area to explore designing self-stabilizing algorithms for overlay networks that are guaranteed to have both small diameter and high expansion. This task is far from trivial as the known non-stabilizing algorithms that satisfy these properties (e.g., [3, 4]) appear to require complicated self-stabilization mechanisms. A desirable scalability property of peer-to-peer networks is low *congestion* — the ability to handle multiple concurrent search requests. Another important property is resistance to *churn* — continuous leaving and joining of nodes. Thus, lowering Tiara's congestion and improving its resistance to churn is a significant avenue of future research.

References

1. Andersen, D., Balakrishnan, H., Kaashoek, F., Morris, R.: Resilient overlay networks. In: SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2001) 131–145
2. Aspnes, J., Shah, G.: Skip graphs. In: SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2003) 384–393
3. Awerbuch, B., Scheideler, C.: The hyperring: a low-congestion deterministic data structure for distributed environments. In: SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2004) 318–327
4. Bhargava, A., Kothapalli, K., Riley, C., Scheideler, C., Thober, M.: Pagoda: a dynamic overlay network for routing, data management, and multicasting. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM (2004) 170–179
5. Harvey, N.J.A., Jones, M.B., Saroiu, S., Theimer, M., Wolman, A.: Skipnet: a scalable overlay network with practical locality properties. In: USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems, Berkeley, CA, USA, USENIX Association (2003) 9–9
6. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: a scalable and dynamic emulation of the butterfly. In: PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing, New York, NY, USA, ACM (2002) 183–192
7. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, ACM (2001) 161–172
8. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Guerraoui, R., ed.: Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings. Volume 2218 of Lecture Notes in Computer Science., Springer (2001) 329–350
9. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. **11**(1) (2003) 17–32
10. Awerbuch, B., Scheideler, C.: Group Spreading: A protocol for provably secure distributed name service. In: Proc. of the 31st International Colloquium on Automata, Languages and Programming (ICALP). (2004)
11. Alima, L.O., Haridi, S., Ghodsi, A., El-Ansary, S., Brand, P.: Position paper: Self-.properties in distributed k-ary structured overlay networks. In: Proceedings of SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems. Volume 3460 of Lecture Notes in Computer Science., Springer (May 2004)
12. Onus, M., Richa, A.W., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: ALENEX 2007: Proceedings of the Workshop on Algorithm Engineering and Experiments, SIAM (January 2007)
13. Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology p2p systems. In: P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing, Washington, DC, USA, IEEE Computer Society (2005) 39–46
14. Hérault, T., Lemarinier, P., Peres, O., Pilard, L., Beauquier, J.: Brief announcement: Self-stabilizing spanning tree algorithm for large scale systems. In: SSS 2006:

Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems. Volume 4280 of Lecture Notes in Computer Science., Springer (November 2006) 574–575
15. Cramer, C., Fuhrmann, T.: Isprp: a message-efficient protocol for initializing structured p2p networks. In: IPCCC 2005: Proceedings of the 24th IEEE International Performance Computing and Communications Conference, IEEE (April 2005) 365–370
16. Caron, E., Desprez, F., Petit, F., Tedeschi, C.: Snap-stabilizing prefix tree for peer-to-peer systems. In Masuzawa, T., Tixeuil, S., eds.: Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings. Volume 4838 of Lecture Notes in Computer Science., Springer (2007) 82–96
17. Bianchi, S., Datta, A., Felber, P., Gradinariu, M.: Stabilizing peer-to-peer spatial filters. In: ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2007) 27
18. Dolev, S., Kat, R.I.: Hypertree for self-stabilizing peer-to-peer systems. Distributed Computing **20**(5) (2008) 375–388
19. Dolev, D., Hoch, E., van Renesse, R.: Self-stabilizing and byzantine-tolerant overlay network. In: OPODIS 2007: Proceedings of the 11th International Conference on the Principles of Distributed Systems. Volume 4878 of Lecture Notes in Computer Science., Springer (December 2007) 343–357
20. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6) (1990) 668–676
21. Munro, J.I., Papadakis, T., Sedgewick, R.: Deterministic skip lists. In: SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1992) 367–375
22. Harvey, N.J.A., Munro, J.I.: Deterministic skipnet. Inf. Process. Lett. **90**(4) (2004) 205–208
23. Fischer, M., Lynch, N., Patterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2) (April 1985) 374–382
24. Chandra, T., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43**(4) (1996) 685–722
25. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Communications of the ACM **43**(2) (1996) 225–267
26. Clouser, T., Nesterenko, M., Scheideler, C.: Tiara: A self-stabilizing deterministic skip list. Technical Report TR-KSU-CS-2008-04, Department of Computer Science, Kent State University (June 2008)